Transact-SQL User-Defined Functions

Andrew Novick

www.NovickSoftware.com

Library of Congress Cataloging-in-Publication Data

Novick, Andrew N.
Transact-SQL user-defined functions / by Andrew Novick.
p. cm.
Includes bibliographical references and index.
ISBN 1-55622-079-0 (pbk.)
1. SQL server. 2. Database management. I. Title.
QA76.9.D3 N695 2003
005.75'85--dc22 2003020942
CIP

©2004, Andrew Novick

All Rights Reserved

38 Cranberry Circle
Sudbury, MA 01776

No part of this book may be reproduced in any form or by any means
without permission in writing from Andrew Novick

Originally published by Wordware Publishing, Plano TX

ISBN 1-55622-079-0

Transact-SQL is a trademark of Sybase, Inc. or its subsidiaries.
SQL Server is a trademark of Microsoft Corporation in the United States and/or other countries.
All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks should not be regarded as intent to infringe on the property of others. The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products.
This book is sold as is, without warranty of any kind, either express or implied, respecting the contents of this book and any disks or programs that may accompany it, including but not limited to implied warranties for the book's quality, performance,merchantability,orfitnessforanyparticularpurpose.Neither Andrew Novick nor Wordware Publishing,Inc. shallbeliabletothepurchaseroranyotherpersonorentitywithrespecttoanyliability,loss,ordamagecaused
or alleged to have been caused directly or indirectly by this book.

Contents

Introduction xv

Part I—Creating and Using User-Defined Functions 1

1 **Overview of User-Defined Functions** 3
Introduction to UDFs 4
Scalar UDFs 4
Inline User-Defined Functions 9
Multistatement Table-Valued User-Defined Functions 12
Why Use UDFs? 16
Reuse 17
Organizing Code through Modularization 19
Ease of Coding 20
Why Not Use UDFs? 21
Performance 21
Summary 22

2 **Scalar UDFs** 23
Creating, Dropping, and Altering Scalar UDFs 25
Permissions to Use CREATE/DROP/ALTER FUNCTION 25
Using the CREATE FUNCTION Statement 28
The Function Body 31
Declaring Local Variables (Including TABLEs) 31
Control-of-flow Statements and Cursors 33
Using SQL DML in Scalar UDFs 34
Adding the WITH Clause 36
Specifying WITH ENCRYPTION 36
Specifying WITH SCHEMABINDING 39
Using Scalar UDFs 44
Granting Permission to Use Scalar UDFs 44
Using Scalar UDFs in SQL DML 45
Using Scalar UDFs in the Select List 45
Using Scalar UDFs in the WHERE and ORDER BY Clauses . . 47
Using Scalar UDFs in the ON Clause of a JOIN 49
Using Scalar UDFs in INSERT, UPDATE, and DELETE Statements 50
Using Scalar UDFs in SET Statements 50
Using Scalar UDFs in EXECUTE and PRINT Statements . . . 51

Using Scalar UDFs in SQL DDL 52
Using Scalar UDFs in CHECK Constraints 53
Using Scalar UDFs in Computed Columns 56
Creating Indexes on Computed Columns with UDFs 58
Summary 61

3 **Working with UDFs in the SQL Server Tools** 63
Query Analyzer 64
Debugging UDFs 65
Creating UDFs with Templates 70
SQL Profiler. 74
Trying to Generate Your Own Trace Events from a UDF. 79
Enterprise Manager and Other Tools. 80
Summary 81

4 **You Can't Do That in a UDF**. 83
Restriction on Invoking Nondeterministic Functions 84
Take the Nondeterministic Value as a Parameter 85
Use the View Trick 86
Restrictions on Data Access 88
Restrictions on the EXECUTE Statement 88
Restriction on Access to Temporary Tables 89
Restriction on Returning Messages 90
Restrictions on Setting Connection Options 91
No SET Commands Allowed 91
The Execution Environment of a UDF 92
Summary 98

5 **Handling Run-time Errors in UDFs** 99
Error Handling Inside UDFs 100
Testing Parameters and Returning NULL 106
Returning Special Values 107
Causing Unrelated Errors in UDFs 108
Reporting Errors in UDFs Out the Back Door 109
Summary. 111

6 **Documentation, Formatting, and Naming Conventions** . . 113
Separator First Formatting 115
Header Comments. 118
* description. 120
* Related Functions 120
* Attribution 120
* Maintenance Notes 120
* Example 120
* Test 121
* Test Script 122

* History . . . 122
* Copyright . . . 122
What's Not in the Header . . . 122
* Parameters . . . 122
* Algorithm and Formulas . . . 123
Naming Conventions . . . 123
Naming User-Defined Functions . . . 123
Naming Columns . . . 129
Domain Names . . . 129
Naming Function Parameters and Local Variables . . . 131
Summary . . . 131

7 **Inline UDFs** . . . 133
Managing Permissions on Inline UDFs . . . 133
Permissions to Create, Alter, and Drop Inline UDFs . . . 134
Permission to Use Inline UDFs . . . 134
Creating Inline UDFs . . . 134
Template for Inline UDFs . . . 136
Creating a Sample Inline UDF . . . 137
Retrieving Data with Inline UDFs . . . 138
Sorting Data in Inline UDFs . . . 140
Updatable Inline UDFs . . . 141
Using Inline UDFs for Paging Web Pages . . . 146
Retrieving Minimal Data for Each Web Page . . . 146
Creating the Inline UDF . . . 147
Summary . . . 153

8 **Multistatement UDFs** . . . 155
Managing Permissions on Multistatement UDFs . . . 156
Permissions to Create, Alter, and Drop Multistatement UDFs . . . 156
Permission to Select on Multistatement UDFs . . . 156
Creating and Using Multistatement UDFs . . . 157
Template for Multistatement UDFs . . . 161
Using Cursors in UDFs with a Template . . . 163
List Management . . . 164
Creating Tables from Delimited Text . . . 165
Turning Tables into Delimited Text . . . 169
Using UDFs to Replace Code Tables . . . 172
Summary . . . 173

9 **Metadata about UDFs** . . . 175
System Stored Procedures to Help with UDFs . . . 176
sp_help . . . 176
sp_help on a Scalar UDF . . . 177
sp_help on an Inline UDF . . . 177
sp_help on Multistatement UDFs . . . 179

sp_helptext 180
sp_rename 181
sp_depends 182
Retrieving Metadata about UDFs 183
Finding out about UDFs in INFORMATION_SCHEMA 183
INFORMATION_SCHEMA.ROUTINES. 184
INFORMATION_SCHEMA.ROUTINE_COLUMNS 185
INFORMATION_SCHEMA.PARAMETERS 185
Built-in Metadata Functions 186
Information about UDFs in System Tables 187
Metadata UDFs 188
Function Information 189
What Are the Columns Returned by a UDF? 190
What Are the Parameters Used When Invoking a UDF? 192
Metadata Functions that Work on All Objects 193
SQL-DMO 196
Summary. 196

10 **Using Extended Stored Procedures in UDFs** 197
Which xp_s Can Be Used in a UDF? 198
xp_logevent. 199
xp_sprintf 208
sp_OA* and OLE Automation: The Keys to the Vault 210
Permissions to Use sp_OA* 210
Picking the Best Uses of OLE Automation 211
Invoking a COM Object 212
Breaking Down an HRESULT 215
Logging OLE Automation Errors 219
Creating a Useful OLE Automation UDF 221
Encapsulating Properties and Methods with UDFs 223
Invoking Your Own OLE Objects 223
Summary. 228

11 **Testing UDFs for Correctness and Performance** 229
Embedding Tests in the Header of the UDF 230
Test Scripts 231
Drilling Down into the Performance Problem 235
The Function and the Template for Testing Performance 236
Constructing a Large Test Table 239
Experimenting with UDF vs. Expression Performance 242
Summary. 244

12 **Converting between Unit Systems** 247
Getting Conversion Factors for the Metric System. 248
Design Issues for Unit Conversions. 250
Scaling the Results 253

Combining Scale and Precision 253
Writing the Conversion Functions 255
Converting Measurements the Simple Way 255
Adding the Choice of Output Units 258
Anything-to-Anything Conversions 260
Putting the Unit Conversion Functions to Work 262
What Is Equal? 266
Testing Numbers for Equality 268
Reducing Bug Reports Due to Numeric Rounding 274
Summary 275

13 **Currency Conversion** **277**
Background on Variable Exchange Rates 278
Design Issues for Currency Conversion 279
Creating the Schema to Support the Functions 279
Picking Data Types for Amounts and Exchange Rates 280
Returning a Meaningful Result When Data Is Missing 281
Writing the Functions for Currency Conversion 283
Using Currency Conversion Functions 289
Summary 291

Part II—System User-Defined Functions 293

14 **Introduction to System UDFs** **295**
Distinguishing System UDFs from Other Functions 296
Naming Requirements for System UDFs 297
Location, Location, Location 297
Referencing System UDFs 298
Pay No Attention to the Man Behind the UDF Curtain 299
Documented System UDFs 300
Special-purpose System UDFs 302
fn_helpcollations 302
fn_virtualservernodes 304
fn_serversharedrives 304
fn_get_sql 305
Viewing a Connection's SQL the Old Way 305
Calling fn_get_sql 306
An Example of Using fn_get_sql 308
Summary 312

15 **Documenting DB Objects with fn_listextendedproperty** . . **313**
Understanding and Creating Extended Properties 314
Maintaining Extended Properties 314
Maintaining MS_Description Using Enterprise Manager 316
Using fn_listextendedproperty 317
The Problem with NULL Arguments to fn_listextendedproperty 320

Understanding NULL Arguments to fn_listextendedproperty . . 322
Task-oriented UDFs that Use fn_listextendedproperty 324
Fetching an Extended Property for All Tables 324
Fetching an Extended Property for All Columns 328
Finding Missing Table Descriptions 331
Reporting on All Tables 333
Summary. 335

16 **Using fn_virtualfilestats to Analyze I/O Performance** . . . 337
Calling fn_virtualfilestats 338
Getting Supporting Information 339
Metadata Functions to Get Database and File IDs. 339
Converting Logical File Names to Physical File Names. 341
What's in sysfiles?. 341
What's in master..sysaltfiles?. 342
How Long Has the Instance Been Up? 344
Statistics for the SQL Server Instance 346
Summarizing the File Statistics with UDFs. 347
Getting Statistics for a Single Database 348
Getting Statistics by Database 349
Breaking Down the I/O Statistics by Physical File. 350
I/O Statistics by Disk Drive 352
Summary. 353

17 **fn_trace_* and How to Create and Monitor System Traces**. 355
Scripting Traces 357
fn_trace_getinfo 360
udf_Trc_InfoTAB 363
Pivoting Data to Create the Columns 364
Mining Bits from the Trace Options Field 366
Stopping Traces. 366
fn_trace_gettable 370
fn_trace_geteventinfo 372
Making a List of Events 374
Additional UDFs Based on fn_trace_geteventinfo. 376
fn_trace_getfilterinfo 377
Converting Operator Codes to Readable Text 377
Sample Output from fn_trace_getfilterinfo 378
Converting the Filter into a WHERE Clause. 379
Reporting on All Running Traces 384
Summary. 386

18 **Undocumented System UDFs** 387
Listing All System UDFs 388
Source Code for the Undocumented System UDFs 389
Selected Undocumented System UDFs. 392

fn_chariswhitespace 392
fn_dblog. 394
fn_mssharedversion 396
fn_replinttobitstring 396
fn_replbitstringtoint 399
fn_replmakestringliteral 403
fn_replquotename 406
fn_varbintohexstr. 409
Summary. 410

19 **Creating a System UDF** 413
Where Do UDFs Get Their Data?. 414
Creating the System UDF. 416
Summary. 419

Appendix A
Deterministic and Nondeterministic Functions 421

Appendix B
Keyboard Shortcuts for Query Analyzer Debugging 427

Appendix C
Implementation Problems in SQL Server 2000 Involving UDFs 429

User-Defined Functions 431

Index. 439

11

Testing UDFs for Correctness and Performance

The first section of this chapter is omitted. It deals with the issue of testing UDFs for correctness.

Drilling Down into the Performance Problem

The problem of inadequate performance is the primary reason not to use scalar UDFs. There can be a big difference in the performance of a query that employs UDFs and one that replaces them with equivalent expressions.

Using UDFs gives your query the same performance as if you implemented the query using a cursor instead of relational SQL. That's because the SQL Server query engine has to address every row and invoke the UDF. It creates a loop very much like a cursor.

Using UDFs is great! Most of the time. But there are times when the extra overhead of UDF execution is more than you can tolerate. Eliminating the UDF, where possible, is a strategy that usually produces dramatically faster query execution times. By "dramatically" I mean two to 500 times faster (enough to make a real difference in the response time of your application).

This section demonstrates and documents the performance problem in detail by constructing an experiment that compares the performance of a UDF-based solution with one that replaces the UDF with an expression. The first step is to pick a suitable UDF for the experiment—one that can be replaced by an expression. Secondly, to make the results dramatic enough, we'll build a table with a sufficient number of rows to show the difference between the UDF and the expression. Finally, we'll run the experiment.

The Function and the Template for Testing Performance

The function that we're going to use is `udf_Txt_CharIndexRev`, which is shown in Listing 11.2. It works just like the built-in `CHARINDEX` function except that it works from the back of the string. Like `CHARINDEX`, the position it reports is relative to the beginning of the string that's searched. I use `udf_Txt_CharIndexRev` for tasks like finding the slash that separates a file name from the rest of a file path.

Listing 11.2: `udf_Txt_CharIndexRev`

```
CREATE FUNCTION dbo.udf_Txt_CharIndexRev (

    @SearchFor varchar(255) -- Sequence to be found
  , @SearchIn varchar(8000) -- The string to be searched
)  RETURNS int -- Position from the back of the string where
               -- @SearchFor is found in @SearchIn
    WITH SCHEMABINDING
/*
* Searches for a string in another string working from the back.
* It reports the position (relative to the front) of the first
* such expression it finds. If the expression is not found, it
* returns zero.
*
* Equivalent Template:
       CASE
         WHEN CHARINDEX(<search_for, varchar(255), ''>
                      , <search_in, varchar(8000), ''>) > 0
         THEN LEN(<search_in, varchar(8000), ''>)
              - CHARINDEX(REVERSE(<search_for, varchar(255), ''>)
                  , REVERSE (<search_in, varchar(8000), ''>)) + 1
         ELSE 0 END
* Example:
select dbo.udf_Txt_CharIndexRev('\', 'C:\temp\abcd.txt')
*
* Test:
PRINT 'Test 1    ' + CASE WHEN 8=
           dbo.udf_Txt_CharIndexRev ('\', 'C:\temp\abcd.txt')
                     THEN 'Worked' ELSE 'ERROR' END
PRINT 'Test 2    ' + CASE WHEN 0=
           dbo.udf_TxtCharIndexRev ('*', 'C:\tmp\d.txt')
                     THEN 'Worked' ELSE 'ERROR' END
****************************************************************/
AS BEGIN

    DECLARE @Result int
          , @StringLen int
          , @ReverseIn varchar(8000)
          , @ReverseFor varchar(255)

    SELECT @ReverseIn = REVERSE (@SearchIn)
         , @ReverseFor = REVERSE(@SearchFor)
         , @StringLen = LEN(@SearchIn)
```

```
    SELECT @Result = CHARINDEX(@ReverseFor, @ReverseIn)

    -- return the position from the front of the string
    IF @Result > 0
        SET @Result = @StringLen - @Result + 1
    -- ENDIF

    RETURN @Result
END
```

Before we get to the experiment, let's try out a few simple cases with this test query:

```
-- Demonstrate udf_Txt_CharIndexRev
SELECT dbo.udf_Txt_CharIndexRev('fdf', 'f123 asdasfdfdfddfjas  ')
            as [Middle]
     , dbo.udf_Txt_CharIndexRev('C', 'C:\temp\ab.txt') as [start]
     , dbo.udf_Txt_CharIndexRev('X', '123456789X') as [end]
     , dbo.udf_Txt_CharIndexRev('AB', '12347') as [missing]

GO

(Results)

Middle      start       end         missing
----------- ----------- ----------- -----------
         13           1          10           0
```

`udf_Txt_CharIndexRev` is simple enough that it's possible to replace it with an expression. The section in the comment block labeled "Equivalent Template" tells how to achieve the same result as the UDF without using the UDF. Here's the template:

```
CASE
  WHEN CHARINDEX(<search_for, varchar(255), ''>
                , <search_in, varchar(8000), ''>) > 0
  THEN LEN(<search_in, varchar(8000), ''>)
       - CHARINDEX(REVERSE(<search_for, varchar(255), ''>)
           , REVERSE (<search_in, varchar(8000), ''>)) + 1
  ELSE 0 END
```

Why bother having the UDF at all if it can be replaced with an expression? As you may recall, my philosophy about writing efficient code is that coding is an economic activity with trade-offs between the cost of writing code and the cost of running it. In my opinion, the best course of action is to write good, easy-to-maintain code and use a basic concern for performance to eliminate any obvious performance problem. The overhead of using a UDF to parse the extension from one file name isn't ever going to show up on the performance radar screen. It's when a UDF is used on large numbers of rows that it becomes a problem. In situations with few rows, having the UDF helps by making the code easier to write.

The Equivalent Template section of the comment block is used to replace a function call to `udf_Txt_CharIndexRev` with the text in the template and then, using SQL Query Analyzer's menu item Edit ➢ Replace Template Parameters, put the function parameters into the revised SQL at the locations where they belong. The Query Analyzer template facility comes in handy because the parameters have to be inserted in more than one place.

So that we don't have to perform the replacement process four times, let's convert this query that does just one call to `udf_Txt_CharIndexRev` into an equivalent query using the template:

```
-- Query to translate
SELECT dbo.udf_Txt_CharIndexRev('fad', 'f123 asdasfdfdfddfjas  ')
            as [Middle]
GO
```

The first step is to replace the function call with the Equivalent Template, and we get:

```
SELECT CASE
         WHEN CHARINDEX(<search_for, varchar(255), ''>
                      , <search_in, varchar(8000), ''>) > 0
         THEN LEN(<search_in, varchar(8000), ''>)
              - CHARINDEX(REVERSE(<search_for, varchar(255), ''>)
                  , REVERSE (<search_in, varchar(8000), ''>)) + 1
         ELSE 0 END
            as [Middle]
```

Next we use the Edit ➢ Replace Template Parameters menu item and enter the two parameters. Figure 11.2 shows the Replace Template Parameters dialog box with the replacement text entered.

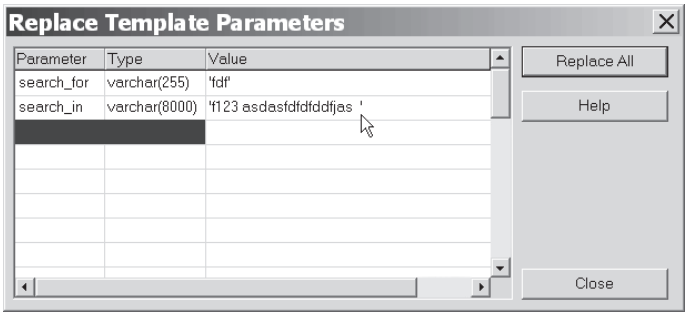

Figure 11.2: Replacing an equivalent template

After pressing the Replace All button, the result looks like this:

```
-- Translated query
SELECT CASE
         WHEN CHARINDEX('fdf'
                     , 'f123 asdasfdfdfddfjas  ') > 0
         THEN LEN('f123 asdasfdfdfddfjas  ')
              - CHARINDEX(REVERSE('fad')
                  , REVERSE ('f123 asdasfdfdfddfjas  ')) + 1
         ELSE 0 END
            as [Middle]
GO

(Results)

Middle
-----------
         13
```

As you can see, replacing the call to the UDF with the equivalent expression results in a much longer and messier query. Simplification, with a corresponding reduction in maintenance effort, is a benefit of using UDFs that is lost when you replace them with complex expressions.

Now that we have a function and can replace it with an expression, we need some data that helps show us the difference in performance between the two. Pubs and Northwind are useful for learning but they don't contain very much data; at best they have a few hundred rows in any one table. I like to run tests on a million-row table. SQL Server doesn't come with any tables that large, so we'll have to construct one.

Constructing a Large Test Table

To test the difference between the UDF and the equivalent expression, we need a table with a large number of rows. Figure 11.3 shows a database diagram for the `ExampleNumberString` table that will be used for the experiment. The `NumberString` column is a 20- to 23-digit numeric string that we can use for experimentation on `udf_Txt_CharIndexRev`.

ExampleNumberString

Column Name	Data Type	Length	Allow Nulls
ID	int	4	
BigNum	numeric	17	✓
NumberString	varchar	128	✓

Figure 11.3: `ExampleNumberString` table fields

The table hasn't been added to your TSQLUDFS database because there is a stored procedure in the database to create the table and populate it with plenty of rows. `usp_CreateExampleNumberString` is shown in Listing

11.3. Its parameter is the number of times to double the number of rows in `ExampleNumberString`.

Listing 11.3: `usp_CreateExampleNumberString`

```
CREATE  PROC usp_CreateExampleNumberString

    @Loops int = 20 -- creates POWER (2, @Loops) Rows
                    -- 20 Creates 1,000,000 rows
AS

    DECLARE @LC int -- Loop counter

    -- delete an existing ExampleNumberString table
    if exists (select * from dbo.sysobjects
                where id = object_id(N'dbo.ExampleNumberString')
                     and OBJECTPROPERTY(id, N'IsUserTable') = 1)
        DROP TABLE dbo.ExampleNumberString

   CREATE TABLE ExampleNumberString (
                ID int identity (1,1)
              , BigNum Numeric (38, 0)
              , NumberString varchar(128) NULL
                )

    INSERT INTO ExampleNumberString  (BigNum, NumberString)
          VALUES(CONVERT (numeric(38,0), rand() * 9999999999999)
                , '                            ') -- preallocate


    SELECT @LC = 0
    WHILE @LC < @Loops BEGIN

        INSERT INTO ExampleNumberString (BigNum, NumberString)
            SELECT BigNum * RAND(@LC + 1) * 2
                  , '                              '
                FROM ExampleNumberString

        SELECT @LC = @LC + 1
    END -- WHILE

   UPDATE ExampleNumberString
        SET NumberString = convert(varchar(128)
                      , convert(numeric(38,0), 9834311) * bignum)
```

Before running `usp_CreateExampleNumberString`, check to be sure you have enough disk space. Adding one million rows takes about 70 megabytes. With the size of today's disks, that shouldn't be a problem, but check anyway. To add the million rows, run the query as is. If you can't spare the 70 megabytes, reduce the parameter from 20 to just 10, which will add only about 1,000 rows. Now create the table:

```
-- Create and populate the table
DECLARE @RC int
EXEC @RC = usp_CreateExampleNumberString 20
GO
```

usp_CreateExampleNumberString does a reasonable job of producing a different large NumberString for each row. There is usually about one percent duplication of numbers, which seems acceptable for most testing scenarios. With a parameter of 20, the procedure takes about 55 seconds to run on my desktop development system.

Since we're going to run multiple queries against the NumberString column to test the time it takes to use a UDF on a large number of rows, we should try to make the circumstances for all queries as similar as possible. One way to do that is to force all the rows that are going to be queried into memory, SQL Server's page cache, and keep them there. If we don't have all the desired rows in memory, then we might be comparing one query that reads a million rows from disk with another query that reads the same million rows from SQL Server's page cache. It wouldn't be a fair comparison.

SQL Server provides the DBCC PINTABLE statement to force all pages from a table to remain in the cache once they've been read the first time. *Use it with caution!* It can fill SQL Server's cache and cause the query engine to lock up to the point where you have to shut it down and restart it. Only do this if you have adequate RAM available, not just virtual memory. There's no point in using virtual memory as a substitute for RAM in this situation. That just substitutes one form of disk I/O (paging) for the one we're trying to eliminate (reading pages from disk).

This next script pins ExampleNumberString in memory. On my desktop development system with a million-row table, pinning the table forced SQL Server to consume 82 additional megabytes of RAM. The system has 512 megabytes of RAM, and I used Task Manager to see that memory isn't full. If your system has less available RAM, reduce the number of rows in the test to eliminate paging.

```
-- Pin ExampleNumberString into memory. You must change the database name, if
-- you're not going to run it in TSQLUDFS
-- Be sure that you have enough memory available before you do
-- this.  My SQL Server Process grew to 82 megabytes when I ran
-- this script.
DECLARE @db_id int, @tbl_id int
SET @db_id = DB_ID('TSQLUDFS') --  DB_ID('<your database name goes here>')
SET @tbl_id = OBJECT_ID('ExampleNumberString')
DBCC PINTABLE (@db_id, @tbl_id)
GO
```

(Results)

```
Warning: Pinning tables should be carefully considered. If a pinned table is larger,
or grows larger, than the available data cache, the server may need to be restarted
and the table unpinned.
DBCC execution completed. If DBCC printed error messages, contact your system
administrator.
```

Pinning the table only tells SQL Server to never remove the table's pages from the page cache. It doesn't read them into the cache. The next query does:

```
-- Read all the rows to force the pages into the cache
SELECT * from ExampleNumberString
GO
```

With a million-row table in memory, the stage is set for comparing the UDF and the equivalent template. Ladies and gentleman, place your bets.

Experimenting with UDF vs. Expression Performance

Let's start by constructing a query that uses `udf_Txt_CharIndexRev` to search for the string `'83'` from the back of the `NumberString` column and then replace the UDF with the equivalent expression. The two queries are:

```
SELECT dbo.udf_Txt_CharIndexRev ('83', NumberString)
                as [Index of 83 from the string end]
    FROM ExampleNumberString

SELECT CASE WHEN CHARINDEX('83', NumberString) > 0
         THEN LEN(NumberString)
              - CHARINDEX(REVERSE('83')
                  , REVERSE (NumberString)) + 1
         ELSE 0 END as [Index of 83 from the string end]
    FROM ExampleNumberString
```

Don't run them yet. I have a few more wrinkles to throw into the experiment.

Any query that returns a million rows to SQL Query Analyzer's results window is going to do a lot of work on sending, receiving, and displaying the results. To eliminate most of that work, using an aggregate function, such as `MAX`, forces the UDF or expression to be evaluated without returning much data as the result.

For comparison purposes, I've also included a third query, labeled Query #0, that just takes the `MAX` of the `LEN` function. This query represents the minimum time it might take to just read the million rows of data.

The `SET STATISTICS TIME ON` command tells SQL Server to measure the time to parse and compile the query and the time to execute the query and report them back with the query results.

Making these three changes to the queries gives us our experiment. If you run them, be patient. Query #1 took over four minutes on my system. The three queries with their results from my desktop development system are:

```
-- Query #0:  The minimum time for an operation that scans the whole table.
SET STATISTICS TIME ON
SELECT MAX(LEN(NumberString)) as [Max Len]
    FROM ExampleNumberString
GO
```

(Results)

```
SQL Server parse and compile time:
   CPU time = 2 ms, elapsed time = 2 ms.Max Length
-----------
         23

SQL Server Execution Times:
   CPU time = 991 ms,  elapsed time = 991 ms.
```

(End of Results for Query #0)

```
-- Query #1: The UDF.  Note execution time is incorrect.
SET STATISTICS TIME ON
SELECT MAX (dbo.udf_Txt_CharIndexRev ('83', NumberString))
             as [Right Most Position]
    FROM ExampleNumberString
GO
```

(Results)

```
SQL Server parse and compile time:
   CPU time = 2 ms, elapsed time = 2 ms.
Right Most Position
-------------------
                 23

SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
```

(End of Results for Query #1 - It took 4 minutes 38 seconds)
(- CPU was working at 100% during that time)

```
-- Query #2: The equivalent expression
SET STATISTICS TIME ON
SELECT MAX (CASE WHEN CHARINDEX('83', NumberString) > 0
         THEN LEN(NumberString) - CHARINDEX(REVERSE('83')
                  , REVERSE (NumberString)) + 1
         ELSE 0 END
        ) [Right Most Position]
    FROM ExampleNumberString
GO
```

(Results)

```
SQL Server parse and compile time:
   CPU time = 3 ms, elapsed time = 3 ms.
Right Most Position
-------------------
                 23

SQL Server Execution Times:
   CPU time = 2934 ms,  elapsed time = 2954 ms.
```

(End of Results for Query #2)

Query #1, the UDF, didn't report its execution times correctly. It seems that `SET STATISTICS TIME` is limited in the duration that it can measure. I used the Windows Task Manager to watch what was happening on my system, and the CPU time for Query #1 is very close to the elapsed time. For our comparison, we'll have to use the elapsed time.

Table 11.1 has the comparison in run time of the queries. The Net column subtracts the time it took to run Query #0, the very simplest expression on the same set of strings, from the other queries. I think that the Net column has the numbers that should be compared because it isolates just the effect of running the UDF or the replacement expression.

Table 11.1: Query timings for the experiment

Query	Description	Time (milliseconds)	Net
#0	Simple expression	991	N/A
#1	UDF	278000	277009
#2	UDF replaced by expression	2954	1963

The difference in time is dramatic. The UDF takes about 140 times longer to run as an equivalent expression. Wow! Four-plus minutes versus three seconds is the kind of difference users really notice.

Of course, the difference isn't going to be perceivable when the query is run on 1, 5, or even 100 rows. Only when the number of rows grows into the thousands does the difference begin to be noticeable.

The lesson that I draw from this experiment is that UDFs must be used with care in performance-sensitive situations. They're a great tool for simplifying code and promoting code reuse, but they can have a dramatically negative effect on performance.

About the Author

Andrew Novick develops applications as consultant, project manager, trainer, and Principal of Novick Software. His firm specializes in implementing solutions to business operations problems using the Microsoft toolset:SQLServer, C#, ASP.NET, and VB.NET. 2007 marks his thirty-sixth year of computer programming, starting in high school with a PDP-8 and moving on to a degree in computer science and an MBA, and then programming mainframes, minicomputers, and for the last 20 years, PCs.

When not programming, Andy enjoys coaching Little League baseball, woodworking, mowing the lawn, and seeing the occasional movie with his wife. He can be found on the web at www.NovickSoftware.com or by e-mail at anovick@NovickSoftware.com. Check out the web site for articles, tips, code samples, and instructional videos about programming SQL Server, VB, C#, ASP.NET, and other environments.